Lecture 5: Support Vector Machine

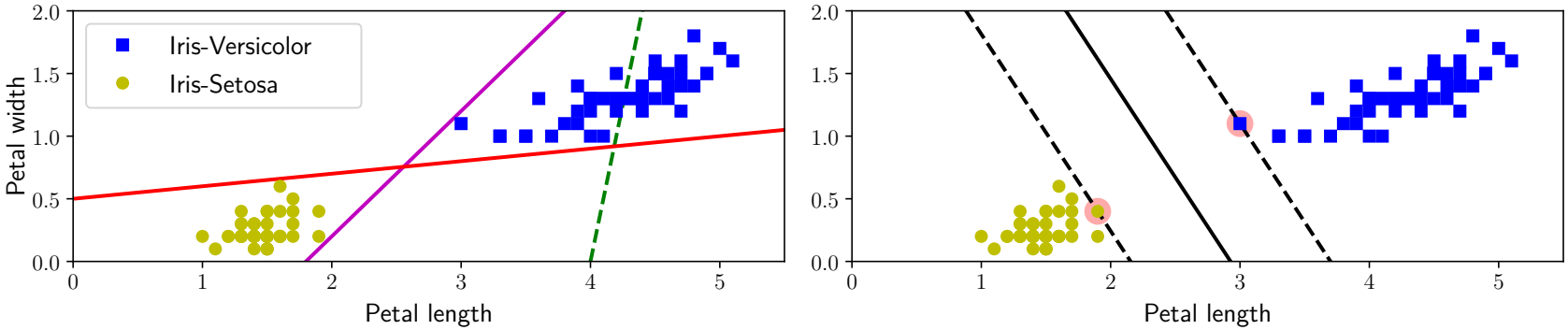
Dr.-Ing. Sudchai Boonto, Assistant Professor

February 24, 2018

Department of Control System and Instrument Engineering, KMUTT

Basic Idea

Linear SVM Classification



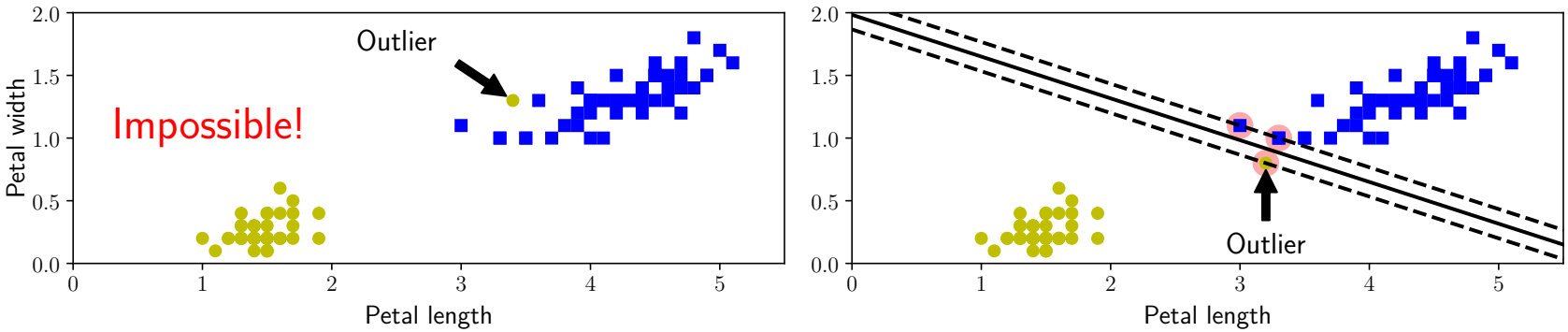
on the left hand side

- the dashed line is so bad
- the other two are ok

on the right hand side

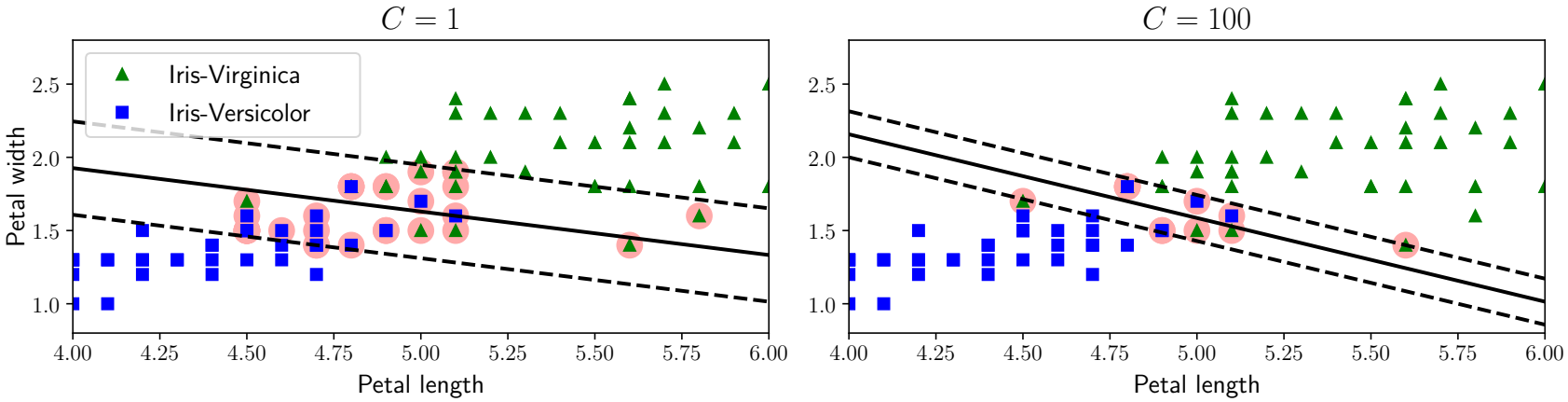
- the solid line in the plot on the right represents the decision boundary of an SVM classifier.
- This line not only separates the two classes but alo stays as far away from the closest training instances as possible.
- This is called **large margin classification.**
- The two dots on the street are called the **support vectors.**

Linear SVM Classification



- If we strictly impose that all instances be off the street and on the right side, this is called **hard margin classification.**
- There are two main issues with hard margin classification. First, it only works if the data is linearly separable, and second it is quite sensitive to outliers.
- the left hand side figure, we have one outlier. It is impossible to find a hard margin.
- On the right hand side we have **Soft margin classification.**
- We allow some margin violations.

Linear SVM Classification



- In Scikit-Learn's SVM classes, we can control this balance using the C hyperparameter.
- a smaller C value leads to a wider street but more margin violations.

Soft Margin Classification

Example how to use Scikit-Learn.

```
from sklearn.svm import SVC
from sklearn import datasets

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

setosa_or_versicolor = (y == 0) | (y == 1)
X = X[setosa_or_versicolor]
y = y[setosa_or_versicolor]

# SVM Classifier model
svm_clf = SVC(kernel="linear", C=float("inf"))
svm_clf.fit(X, y)
w = svm_clf.coef_[0]
b = svm_clf.intercept_[0]
print(w, b)
```

Note:

$$w_0 x_0 + w_1 x_1 + b = 0 \Rightarrow x_1 = -\frac{w_0}{w_1} x_0 - \frac{b}{w_1}$$

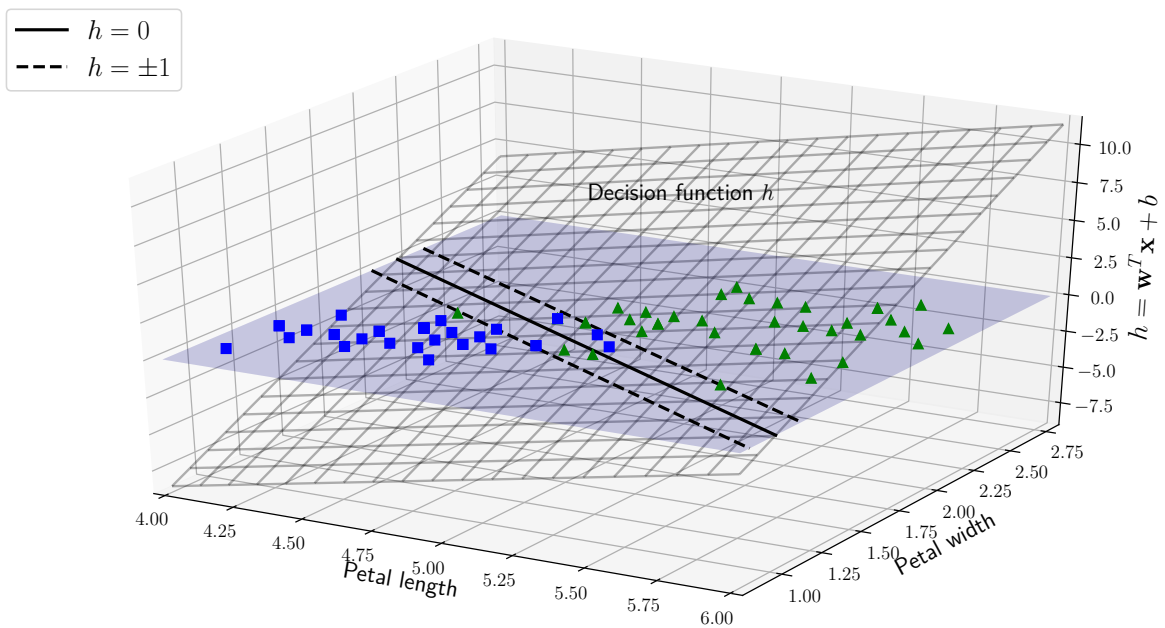
Mathematic Background

The linear SVM classifier model predicts the class of a new instance x by simply computing the decision function

$$w^T x + b = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

where

$$\hat{y} = \begin{cases} 0 & , w^T x + b < 0 \\ 1 & , w^T x + b \geq 0 \end{cases}$$



Mathematic Background

The optimization problem for the Hard margin linear SVM classifier objective

$$\begin{aligned} \underset{w,b}{\text{minimize}} \quad & \frac{1}{2}\|w\|^2 \\ \text{subject to} \quad & t^{(i)}\left(w^T x^{(i)} + b\right) \geq 1,\ i = 1, 2, \cdots, m, \end{aligned}$$

where $t^{(i)} = -1$ for negative instances (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$). This is from the scaling of w and b are not effected $t^{(i)}$.
The optimization problem for the Soft margin linear SVM classifier objective

$$\begin{aligned} \underset{w,b}{\text{minimize}} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\zeta^{(i)} \\ \text{subject to} \quad & t^{(i)}\left(w^T x^{(i)} + b\right) \geq 1 - \zeta^{(i)},\ i = 1, 2, \cdots, m \\ & \zeta^{(i)} \geq 0,\ i = 1, 2, \cdots, m \end{aligned}$$

Dual Problem

The optimization problem for the Hard margin linear SVM classifier objective

$$\begin{aligned} \underset{w,b}{\text{minimize}} \quad & \frac{1}{2}\|w\|^2 \\ \text{subject to} \quad & t^{(i)}\left(w^T x^{(i)} + b\right) \geq 1, \; i = 1, 2, \cdots, m. \end{aligned}$$

The constraints can be written as

$$g_i(w) = -t^{(i)}(w^T x^{(i)} + b) + 1 \leq 0$$

The Lagrangian for the optimization problem is

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{m} \alpha_i \left[t^{(i)}(w^T x^{(i)} + b) - 1\right]$$

Note that there are only α_i but no β_i Lagrange multipliers, since th eproblem has only inequality constraints.

Dual Problem

To find the dual form of the problem:

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{m} \alpha_i t^{(i)} x^{(i)} = 0 \implies w = \sum_{i=1}^{m} \alpha_i t^{(i)} x^{(i)}$$

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i t^{(i)} = 0$$

Substitute the results to the Lagrangian, and simplify, we get

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} t^{(i)} t^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)} - b \sum_{i=1}^{m} \alpha_i t^{(i)}$$

All together we have

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} (x^{(i)})^T x^{(j)} - \sum_{i=1}^{m} \alpha^{(i)} \\ \text{subject to} \quad & \alpha^{(i)} \geq 0, \ i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i t^{(i)} = 0 \end{aligned}$$

Dual Problem

One you find the vector $\hat{\alpha}$ that minimizes the dual optimization we have

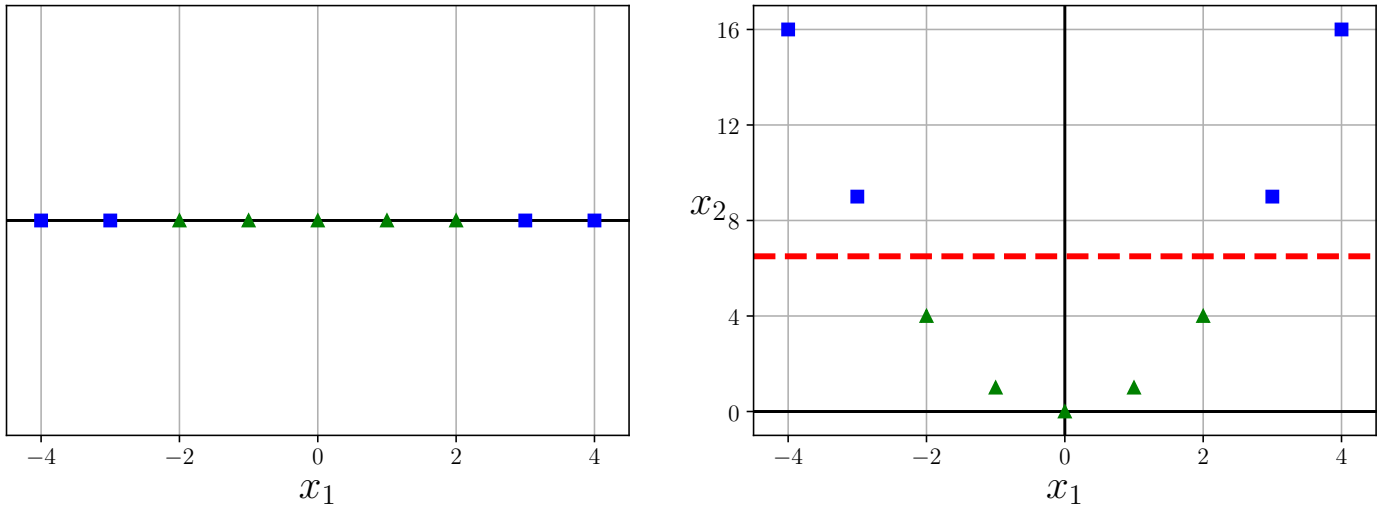
$$\hat{w} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} x^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{i=1}^{m} \left(1 - t^{(i)} \left(\hat{w}^T x^{(i)} \right) \right), \ \hat{\alpha}^{(i)} > 0$$

- The dual problem is faster to solve than the primal when the number of training instances is smaller than the number of features.
- It makes the kernel trick possible, while the primal does not.

Nonlinear

Nonlinear SVM Classification



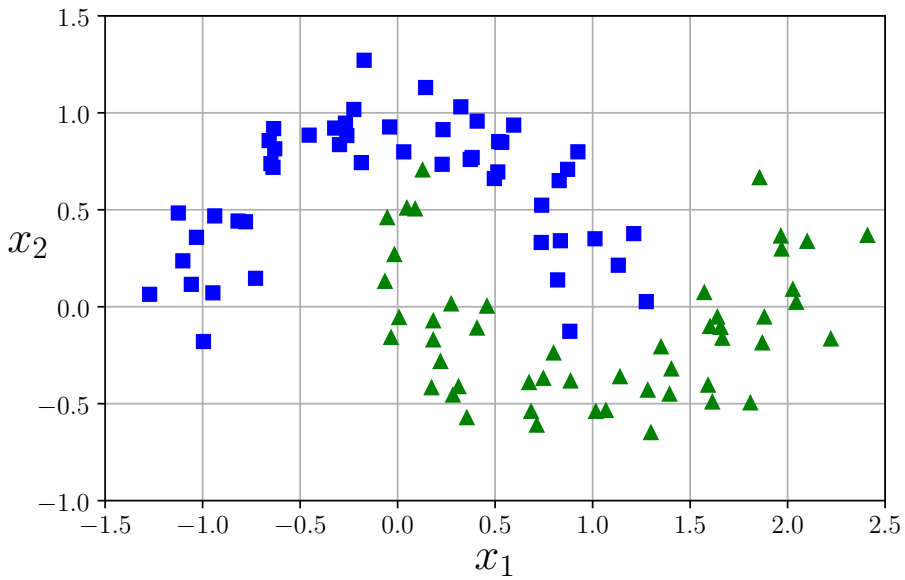
- the left hand side, it represents a simple dataset with just one feature x_1. This dataset is not linearly separable, as you can see.
- By adding the second feature $x_2 = (x_1)^2$, the resulting 2D dataset is linearly separable.

Moon Dataset

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
```



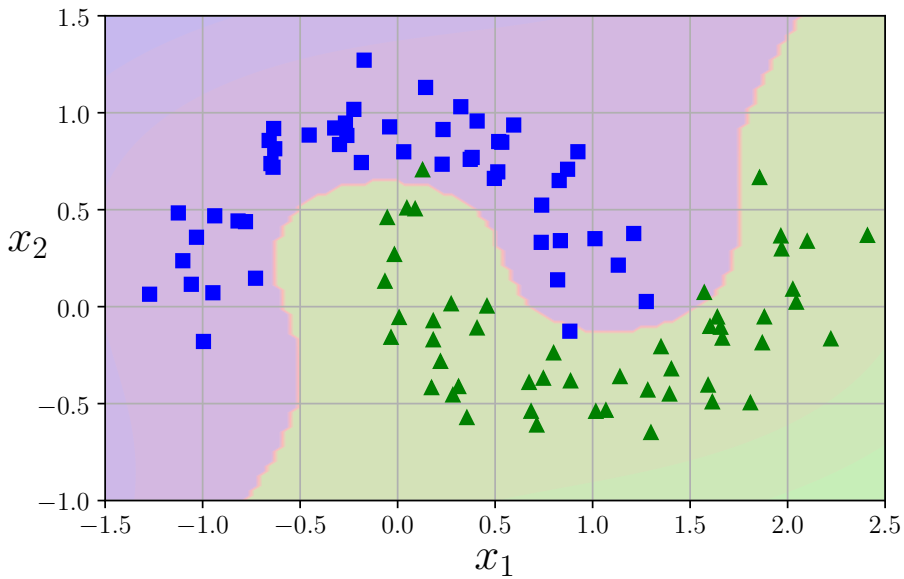
It is not linear separable.

Polynomial Regression

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.svm import LinearSVC

polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge", random_state=42))
    ])

polynomial_svm_clf.fit(X, y)
```



Using Pipeline containing a `PolynomialFeatures`, `StandardScaler`, and `LinearSVC`.

Polynomial Regression

To plot the separate line:

```
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])
```

Polynomial Kernel

Adding polynomial features is simple to implement and can work with all sorts of Machine Learning Algorithms.

- However at a low polynomial degree it cannot deal with very complex datasets,
- With a high polynomial degree it creates a huge number of features, making the model too slow.

Second-degree polynomial mapping

$$\phi(x) = \phi\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}$$

Kernel trick for a 2^{nd}-degree polynomial mapping (vector a and b)

$$\phi(a)^T\phi(b) = \begin{bmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{bmatrix}^T \begin{bmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{bmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_1^2b_2^2$$

Polynomial Kernel

$$a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_1^2 b_2^2 = (a_1 b_1 + a_2 b_2)^2 = \left[\begin{bmatrix} a_1 \\ a_2 \end{bmatrix}^T \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right]^2 = (a^T b)^2$$

$$\phi(a)^T \phi(b) = (a^T b)^2$$

The dot product of the transformed vectors i equal to the square of the dot product of th eoriginal vectors

$$\phi(a)^T \phi(b) = (a^T b)^2$$

- If we transform all training instance with $\phi(x^{(i)})$, the dual problem will contain the dot product $\phi(x^{(i)})^T \phi(x^{(i)})$
- But if ϕ is the 2^{nd}-degree polynomial transformation as before, the we can replace the dot product of transformed vectors simply by $\left((x^{(i)})^T x^{(j)}\right)$
- So we don't need to transform the training instances: just replace the dot product by its square.

Polynomial Kernel

Kernel function of the polynomial kernel is

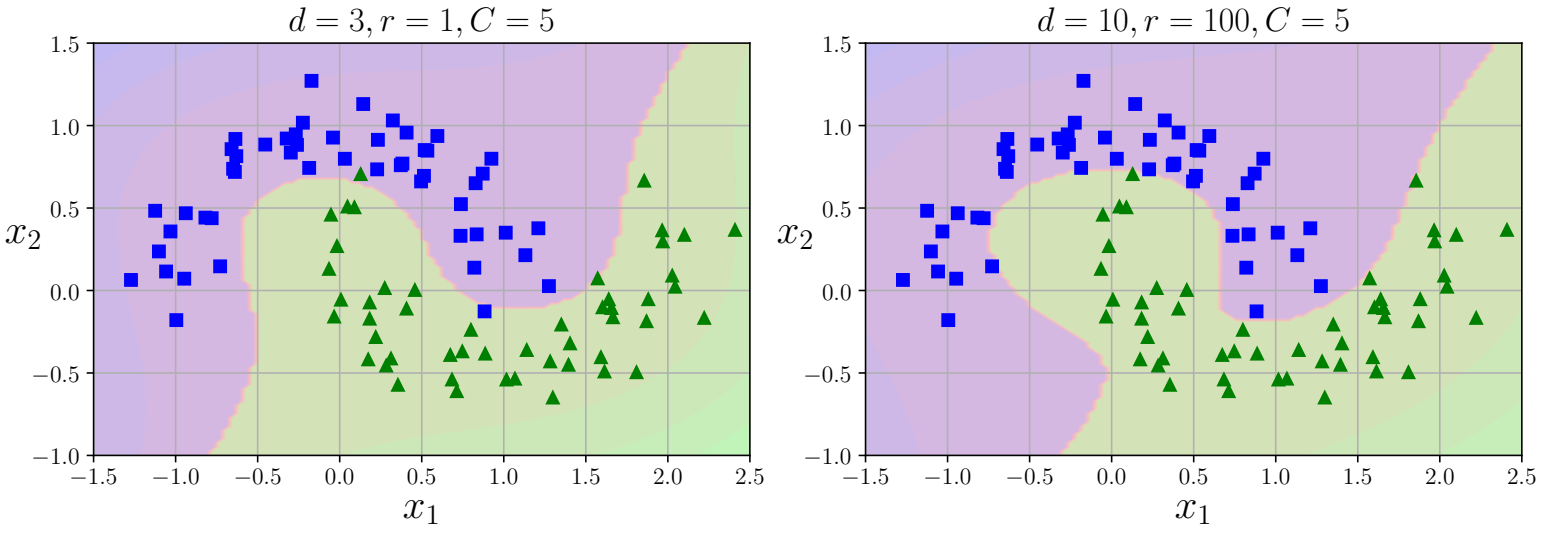
$$K(a, b) = \left(\gamma a^T b + r\right)^d$$

where d is a degree, r is the hyperparameter `coef0` controls how much the model is influenced by high degree polynomials versus low-degree polynomials. (`gamma` = 1)

```
from sklearn.svm import SVC

poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

Polynomial Kernel



A common approach to find the right hyperparameter values is to use grid search. It is often faster to first do a very coarse grid search, then a finer grid search around the vest values found.

Gaussian Radial Basis Function

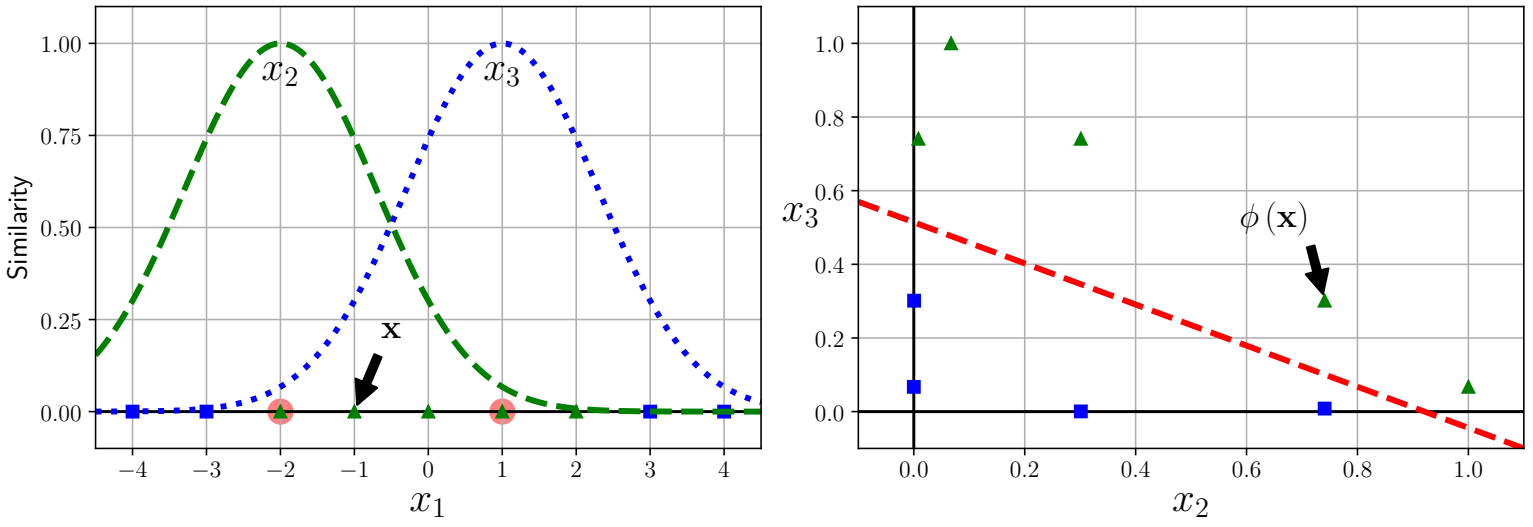
Another technique to tackle nonlinear problems is o add features computed using a **similarity function** that measures how much each instance resembles a particular landmark. For example

- Let consider the one-dimensional dataset and add two landmarks to it a $x_1 = -2$ and $x_1 = 1$
- Define the similarity function to be **Gaussian Radial Basis Function (RBF)** with $\gamma = 0.3$

$$\phi\gamma(x, l) = e^{-\gamma\|x-l\|^2}$$

- For example, $x_1 = -1$ located at a distance of 1 from the first landmark, and 2 from the second landmark. Therefore its new features are $x_2 = e^{(-0.3\times 1^2)} \approx 0.74$ and $x_3 = e^{(-0.3\times 2^2)} \approx 0.30$

Gaussian Radial Basis Function



- Now it is linearly separable by using the transform method.
- It should be trad off between creating all landmarks for all features and the speed of the computation.
- A training set with m instances and n features gets transformed into a training set with m instances and m features (assuming you drop the original features). If your training set is very large, you end up with an equally large number of features.

Gaussian RBF Kernel

To reduce the computational time to compute all the additional feature, we can use the kernel trick

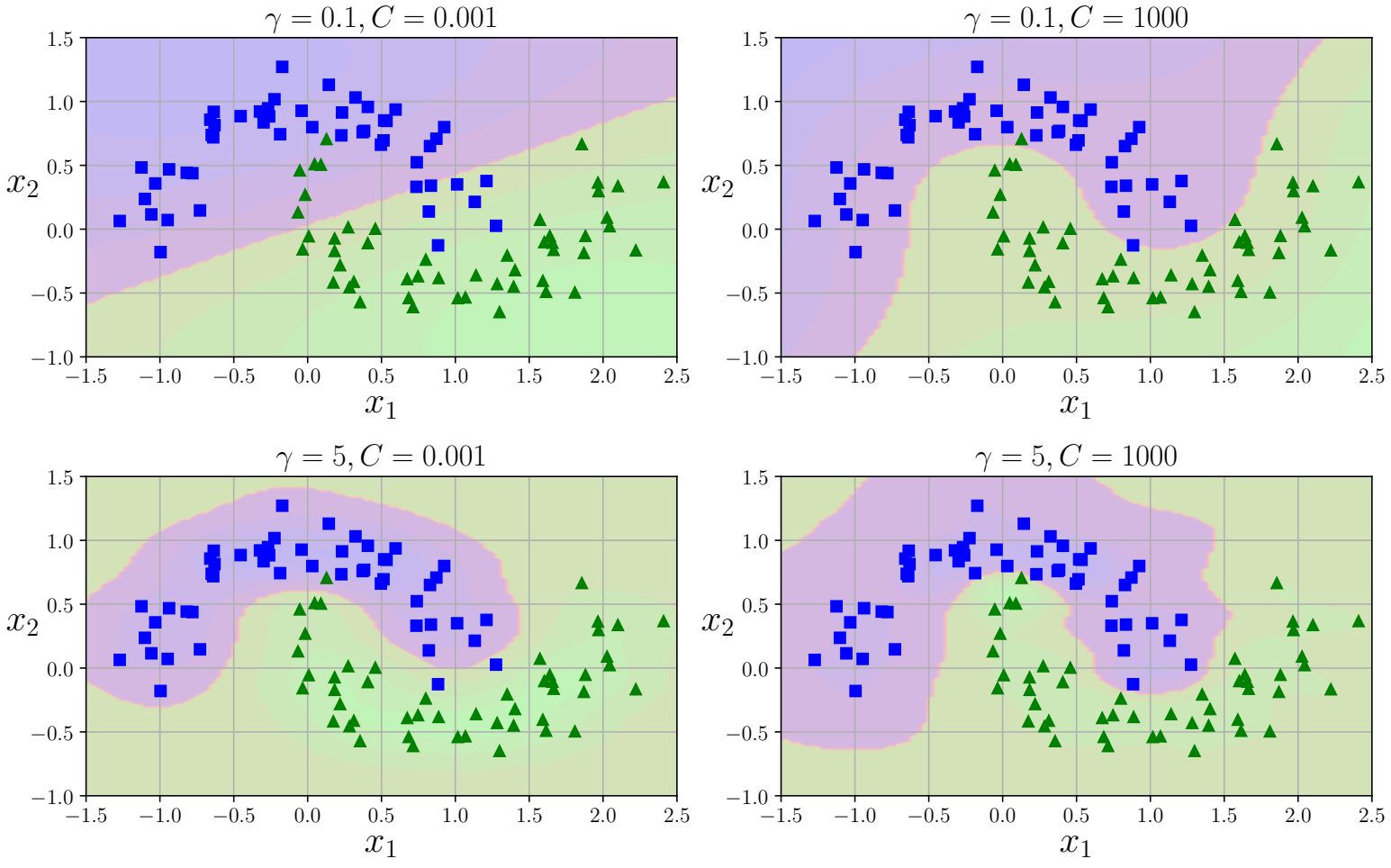
$$K(a, b) = e^{-\gamma \|a-b\|^2}$$

The code for using the Gaussian RBF kernel using the SVC class:

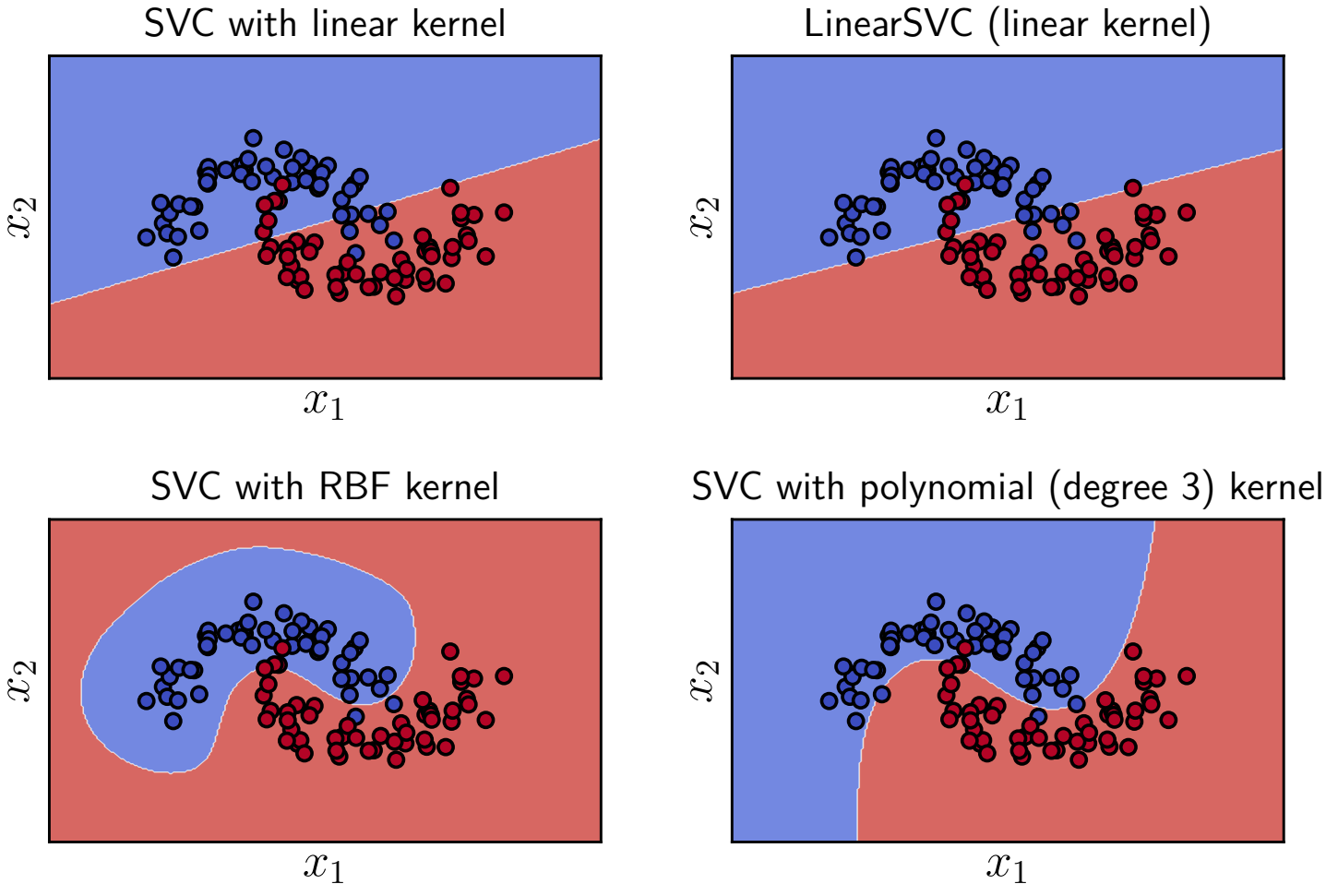
```
rbf_kernel_svm_clf = Pipeline((
            ("scaler", Standardcaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ))
    rbf_kernel_svm_clf.fit(X,y)
```

- Increasing γ makes the bell-shape curve narrower, and as a result each instance's range of influence is smaller: the decision boundary ends up being more irregular; wiggling around individual instances.
- a small γ values makes the bell-shaped curve wider, so instances have a larger range of influence, and the decision boundary ends up smoother.
- Here the γ and C act like regularization hyperparameters: if you model is overfitting, you should reduce them, and if it is underfitting, you should increase them.

Gaussian RBF Kernel



Comparison



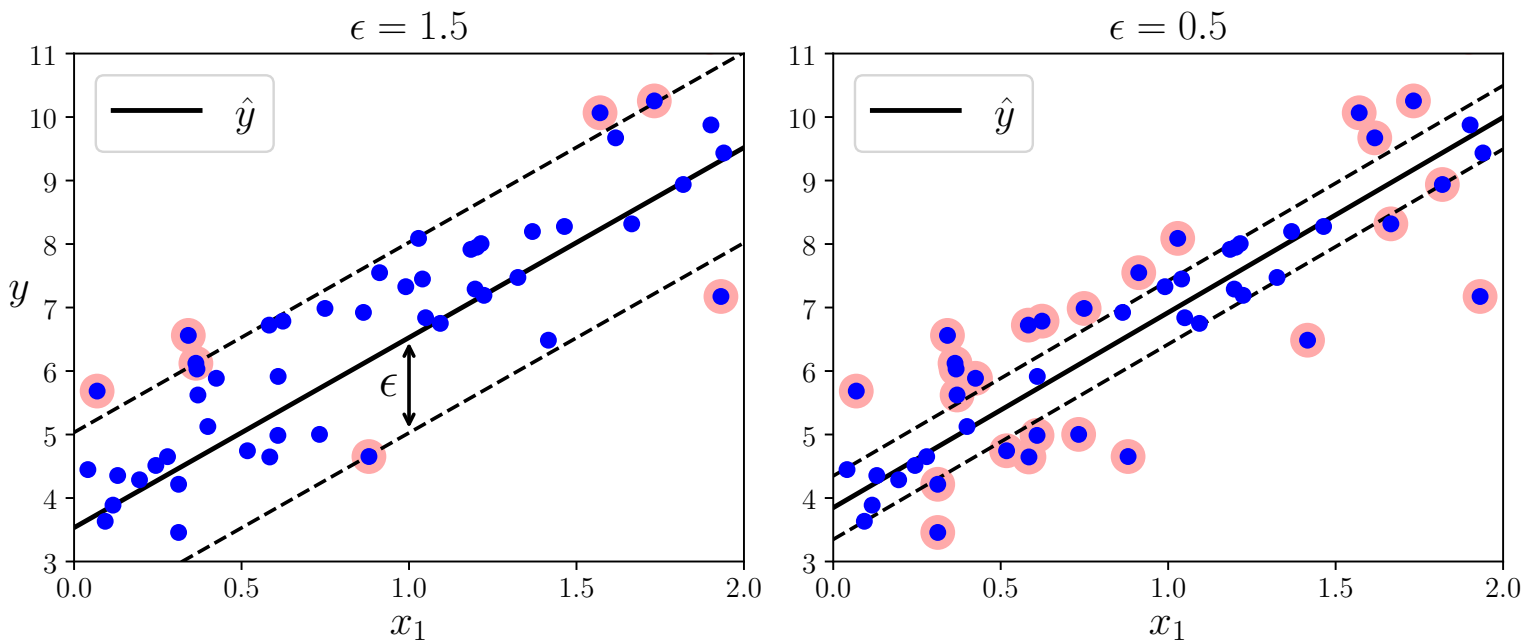
Note `LinearSVC` is much faster than `SVC(kernel="linear")`, but it does not support the Kernel trick.

SVM Regression

The SVM can solve a regression problem as well.

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X,y)
```



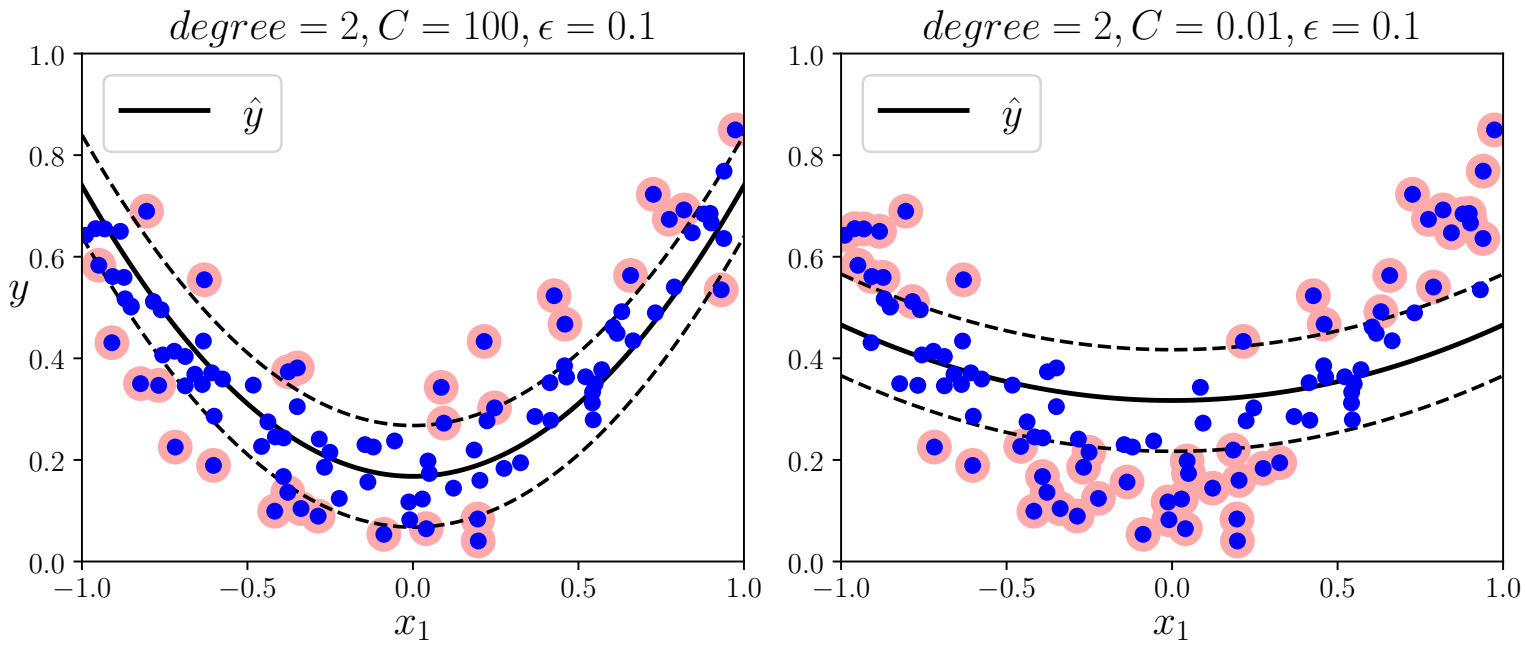
- The SVM Regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instance off the street.)
- The width of the street is controlled by a hyperparameter ε .

Nonlinear SVM Regression

For nonlinear problem:

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X,y)
```



The smaller C value, the greater regularization.

Reference

1. Sebatian Ruder, "*An Overview of Gradient Descent Optimization Algorithms*", arXiv:1609.04747v2, 2017
2. Aurélien, Géron, "*Hands-On Machine Learning with Scikit-Learn & TensorFlow*", O'reilly, 2017
3. S.P.K. Spielberg, R. B. Gopaluni, P. D. Loewen, "Deep Reinforcement Learning Approaches for Process Control", 2017 6th International Symposium on Advanced control of Industrial Processes (AdCONIP), May 28-31, 2017, Taipei, Taiwan
4. Andrew Ng, "*Support Vector Machines*", CS229 Lecture Note
5. Sckit-Learn website `http://scikit-learn.org/stable/index.html`